GPU Accelerated Neural Networks With Dynamic Topology

Travis Chung Shashank Golla Suhib Sam Kiswani

April 24, 2015

Abstract

Neural Networks lend themselves naturally to being parallelized, although certain caveats exist. Furthermore, there exists little work on the benefits of parallelizing networks having a dynamic topology. We demonstrate a parallel implemention of the the Synfire-Growth simulation (developed by Jun and Jin [1]) that runs on the GPU. Despite the dynamic topology of the neural network, our results show that porting the work of [1] to the GPU provided useful runtime performance benefits.

1 Introduction

The human brain is one of the most complicated and interesting biological systems in nature. Despite our poor understanding of the brain's complex structure, it is common to describe it as a biological computer, consisting of an dense and tangled network of neurons, which serve as pathways for electrical current. This analogy provides a useful starting point for computational models, and is often employed in the machine learning domain.

Neural Networks (or NN's) can be understood most simply as a complete weighted graph, with vertices representing neurons, and weighted edges representing the strength of a neural connection. This model can be enhanced by allowing the weights to change in response to external simuli, representing a dynamic topology of connected neurons.

This paper describes the simulation of neural networks with dynamic topology on the GPU using the model developed by [1] in 2007. We describe how the implementation of [1] was refactored to run on the GPU and take advantage of parallelization. Our work concludes with a cost-benefit analysis of running this simulation on the GPU, and how neural networks naturally lend themselves to the parallel environment.

2 Background

Primarily, the study the neuroscience is concerned with the behavior of the brain, the way it is structured, the way it learns, how it develops, how it adapts, and changes with respect to stimuli. While understanding of the central nervous system continues to grow, there is so much that is unknown about the brain: the storage and accessing of memories, how the brain retains information, the

idea of consciousness, and how sensory input is translated into smell, taste, or pain. The modeling and simulating of brain activity is vital in observing the behavior of the brain and building intuition.

There are generally two families of algorithms for the simulation of neural networks, described in detail by [2]. The two families are synchronous ("clock-driven") or asynchronous ("event-driven") algorithms. In synchronous algorithms neurons are updated only when they receive or emit a spike, whereas "clock-driven" algorithms update all neurons simultaneously at every tick of a clock. There are plenty of simulations using synchronous algorithms, because the spike times are a defined time grid. To get exact simulations of neuron spiking, asynchronous algorithms are recommended. Asynchronous algorithms have been developed for simpler models, but for very large networks, the simulation time and number of spikes becomes problematic for computation.

3 Previous Work

Despite the group's lack of domain knowledge regarding neural networks, the work of [3] provided the initial foundation and direction for our work. The comparisons of the Hodgkin-Huxel (HH) and Izhikevich models of neural networks provided by [3] was a significant aide in deciding which model to use for our simulation. In addition to their analysis, [3] provides evidence for parallel implementations on the GPU that achieve speedups of greater than 110 times their corresponding CPU implementations. Though we could not reproduce a speedup of their magnitude, it gave our work its initial direction.

The model we chose to implement on the GPU was one developed and detailed by [1], suggested by an informal advisor for this project.

In addition to developing the simulation model, it was also implemented by [1]. By using their implementation as our foundation, we were able to run a faithful representation of their work on the GPU which provides a convenient metric for the efficacy of our work.

4 Implementation Details

Algorithm 4.1 Synfire Growth Trial

```
while t < TRIAL_TIME do
    MembranePotential(dt)
    for n ∈ N that spiked do                          ▷ Spike Loop
        Check to see if spiking neuron is saturated.
        If spiking neuron isn't saturated, send spikes along active connections.
        Perform Spike Timing Dependent Plasticity on excitatory synapses.
    end for
    for n ∈ N that spiked do                          ▷ Inhibition Phase
        Calculate inhibition.
    end for
    t ← t + dt
end while
SynapticDecay()
```

Most of the complexity in Algorithm 4.1 lies in updating the membrane potential layer, which corresponds to integrating the potential of each neuron at time t over the timestep, dt; i.e. integrating each neuron from $t \rightarrow t+dt$. The calculation is relatively intensive, and is done per neuron in the network. The calculation is a leaky integration model used by [1], and is defined as:

$$\tau_m \frac{dV_m(t)}{dt} = (E_l - V_m) - g_{exc}(t)V_m(t) + g_{inh}(t)(E_{inh} - V_m(t))$$

Where τ_m = 20ms is the membrane time constant; V_m is the membrane potential in millivolts, E_l = −85mV is the leak reversal potential; $g_{exc}(t)$ is the exictatory conductance due to all excitatory synapses; $g_{inh}(t)$ is the inhibitory conductance from all inhibitory synapses; E_{inh} = −75mV is the reversal potential of the inhibitory synapses.

The "Spike Loop" of Algorithm 4.1 is the process of applying long term potentiation (LTP) to incoming synapses and long term depression (LTD) to outgoing synapses, as defined by [1]. This involves strengthing and weakening synapses based on a their spike times within a given window, Δt. Our implementation defaults this window to Δt = 200ms.

Another significant runtime calculation involved calculating the synaptic decay after each trial, which involves an iteration over every synapse in the network (which is of size $O(n^2)$) and applying an expontential decay to the strength of the synapse.

4.1 On the CPU

A significant part of the development time required creating a competitive implementation on the CPU which required a total rewrite of the most current Synfire Growth implementation. The most important detail of the rewrite was pulling out the details of an individual trail (Algorithm 4.1) and modularizing the high level steps, which would allow a individual phases to be run on a GPU kernel independently of other phases.

The CPU code was not as performant as it could've been, and the rewrite resulted in an implementation that would be competitive with the GPU code, and thus provide a standard with which to measure the effect the GPU had on performance. One of the most significant of these CPU optimizations was doing away with the multiple memory accesses and pre-allocating memory buffers before running the trials, instead of allocating during a trial. Due to the fact that the original implementation will end up using the same amount of memory by the end of every trial, this provided a runtime performance boost at no cost.

With dedicated memory buffers, the dependencies between phases could be established and the bandwidth between the host and device could be minimized by excluding unnecessary memory transfers.

Original Synaptic Decay: 10 Trials

Network Size	**Trial Average**	**Avg. Timestep**
200	6.961 ms	2.494 ms
1000	34697.2 ms	966.5 ms
2000	70110.2 ms	18.855 ms

Improved Synaptic Decay: 10 Trials

Network Size	Trial Average	Avg. Timestep
200	1824.62 ms	0.511 ms
1000	8242.2 ms	1.984 ms
2000	17908.5 ms	4.5725 ms

4.2 On the GPU

Given the heavy runtime cost associated with integrating the membrane potential layer and calculating the synaptic decay of the network, they stood out as the things that should be ported to the GPU.

The most significant difficulty in a parallel implementation of the SynfireGrowth was a constraint on random number generation. We chose to remain faithful to the original implementation in providing deterministic evolution of the network, which requires that the occurence of spontaneous activity needed to be deterministic. Our solution was to generate the sequence of numbers each neuron would need on the CPU, and then copying over this sequence to the GPU when required. Naturally this increased the bandwidth between the host and device which implied that networks that were significantly large might incur a significant memory transfer cost.

4.2.1 Membrane Potential Update Kernel

The functionality of the Membrane Potential Loop (MembranePotential(dt) in Algorithm 4.1) is to update each neuron in the network if the neuron has spiked. It calls a function called Update in order to update the whospike array which keeps track of exactly which neurons have spiked by index. In the original, the whospike array was an int array, that when indexed contained each of the indexes of Neurons that have spiked. We have instead made this a boolean array and the "Update" function instead turns the index into a 1 if the corresponding Neuron has spiked.

By transferring the Membrane potential loop onto the GPU, we have used each thread generated by the kernel to update individual Neurons. We have also greatly reduced overhead by making our whospike array a boolean and not having to allocate and deallocate the integer array, as it was in the source code. The kerenel function still calls the Update function in order to update the whospike array, in this function another computationally intensive function called neur_dyn is called.

The original implementation of neur_dyn function utilized four arrays, all of size 3, to host the results of each progression of the function and eventually update the membrane potential voltages, excitation conductance, and inhibitory conductance. While the array address is stored in the local register, the contents of the arrays are stored in global memory, therefore, each call to an array required roughly 500 clock cycles to execute. Overall, this implementation accessed global memory 92 times; which in turn took 46000 clock cycles.

To reduce global memory access, we utilized three local registers to temporarily hold the membrane potential and conductance's and 3 local registers, in place of the arrays, each holding an object of type double, to store the results of each progression. After each progression, the temporary voltages and conductance's are updated. By replacing the arrays with single variables, we had

accessed global memory only 3 times. We also eliminated the instance of thread divergence by utilizing the additive identity of the spike voltage. While this required 8 extra floating point operations, the performance benefits of parallel execution, across all threads, far exceeds the extra overhead.

4.2.2 Synaptic Decay Kernel

Algorithm 4.2 Synaptic Decay

```
for idx < NETWORK_SIZE^2 do
    CheckThreshold(syn_str, idx, syn_type, pd_type)
    CheckThreshold(syn_str, idx, syn_type, pd_type)
    Reduce Synaptic Strength
end for
```

Algorithm 4.3 Check Threshold

```
if syn_type = Active then
    Set Active threshold
else if syn_type = Super then
    Set Super threshold
end if
if pd_type = Potentiation then
    Activate(syn_type, idx)
else if pd_type = Depression then
    Deactivate(syn_type, idx)
end if
```

Algorithm 4.4 Deactivate

```
if syn_type = Active then
    Decrement Active Synapse counter
    Flag index false
else if syn_type = Super then
    Decrement Super Synapse counter
    Flag index false
end if
```

The Synaptic Decay function required the most time to execute and produced the most overhead per function call. This model we had used required, per trial, the signaling of an inhibitory response through all active and super synaptic connections. This was executed by reducing each synaptic strength measurement with a constant synaptic decay factor. During this process, if the decayed, synaptic strength of each connection failed to exceed the corresponding active or super synaptic threshold, the connection to the post synaptic neuron would be trimmed, and no longer present in either the active or super synaptic network.

The original implementation created two, two-dimensional int arrays, composed of heap memory, to host the post-neuron index of the corresponding active or super synaptic connection. Moreover, the rows corresponded to the index of

the pre-neuron, while the column was a pointer to an array of post neurons. When an active or super synaptic connection needed to be trimmed and eliminated from the corresponding network, the program required the allocation of a new array, to house the smaller array of connections, the transfer of elements from the old to the newly allocated array, and the deallocation of the obsolete, heap memory. The constant allocation and deallocation for each active and super synaptic array, per neuron resulted in much overhead. Naturally, since the number of the active and super synaptic structures, for each neuron is different, the implementation is ideally, not suited for execution on the GPU.

To efficiently run the Synaptic Decay model on the GPU, we decided to restructure the Synapse class by creating two static Boolean arrays to house the active and super synaptic connections. Each array was a complete network, of size, $network\_size \cdot network\_size$, where $network\_size$ is the number of neurons existing in the network. If there existed an active or super synaptic connection from one neuron to another neuron, the corresponding index in the corresponding complete network would be flagged as true.

This new implementation is much more suited for execution on the GPU. We greatly reduced the overhead of having to allocate and deallocate the array structure per neuron, by replacing this implementation with one that simply flags the corresponding synaptic connection. Furthermore, since each thread is only required to flag its corresponding thread index, we are able to utilize the Single instruction, multiple thread (SIMT) model, eliminate thread divergence, and eliminate unnecessary work load.

5 Results

Below is the table of the execution times across the restructured CPU code, GPU code, and the original source code. The restructured Synapse class, which utilized two static Boolean arrays to house the active and super synaptic connections, resulted in a speed up of 430x over the original implementation, which required the constant allocation and deallocation of each active and super synaptic array, per neuron. Naturally, this restructured implementation is well suited for execution on the GPU, resulting in a speed up 5.46x over the restructured CPU code, and speed up of 2298.9x over the original implementation.

Synaptic Decay: Execution times	
Platform	Time (μs)
GPU	0.4126
CPU	2.2529
Original	969.2

The restructured CPU code is naturally more suited for execution on the GPU. We greatly reduced the overhead of having to allocate and deallocate the array structure per neuron, by replacing this implementation with one that simply flags the corresponding synaptic connection. Since each thread is only required to flag its corresponding thread index, we are able to utilize the Single instruction, multiple thread (SIMT) model, eliminate thread divergence within the Deactivate function, and eliminate the unnecessary work load.

While the new implementation provides considerable speed up over the original implementation, this new approach is limited on the size of the network.

Since the size of our network was 1000, the GPU only required 32 MB of space, 16 MB per active and super synaptic network. If the network size was increased to 10,000, then the GPU would require 3.2 GB of space. Therefore, the larger the network, more space of global memory is required on the GPU. If space is exceeded, extra overhead will be required to copy data from host to device and back from device to host.

6 Future Work

The eventual goal is to run an entire trial on the GPU device. A constraint on our implementation was deterministic behavior of the neural network (e.g. a network can be recreated given the initial seed and training neurons). The inability to determine an optimal method to generating a deterministic sequence of random numbers on the device proved to be a limitation, although a non-deterministic implementation could potentially be useful as it would be able to run exclusively on the GPU.

Another limitation on running the entire network on the GPU is the fact that synaptic strengths need to remain global synchronized across all threads. This makes a concurrent implementation difficult as it introduces competition between threads when updating the synapses between two neurons throughout the various phases. The natural advantage of an entire trial running on the GPU is the removal of the memory transfer overhead for each timestep.

References

[1] Jun, Joseph K and Jin, Dezhe Z, *Development of neural circuitry for precise temporal sequences through spontaneous activity, axon remodeling, and synaptic plasticity.* PLoS ONE (2007) 2(8): e723. doi: 10.1371/journal.pone.0000723

[2] Brette R, Rudolph M, Carnevale T, et al. *Simulation of networks of spiking neurons: A review of tools and strategies. Journal of computational neuroscience.* 2007;23(3):349-398. doi:10.1007/s10827-007-0038-6.

[3] Fidjeland, A.K., Shanahan, M.P. *Accelerated simulation of spiking neural networks using GPUs.* The 2010 International Joint Conference on Neural Networks (IJCNN). (2010)